Programming Storage-centric Sensor Networks with Squirrel

Luca Mottola
Swedish Institute of Computer Science
luca@sics.se

ABSTRACT

We present SQUIRREL, a stream-oriented programming framework for storage-centric sensor networks. The storage-centric paradigm—where storage operations prevail over communication activity—applies to scenarios such as batch data collection, delay-tolerant mobile applications, and disconnected operations in static networks. SQUIRREL simplifies developing such applications by decoupling data processing from storage, and by transparently handling the latter. We achieve this through: *i)* a modular programming abstraction, and *ii)* a lightweight run-time layer that efficiently allocates data to different storage areas, based on size vs. energy trade-offs. We demonstrate SQUIRREL's effectiveness based on three real-world applications, each representing a different storage-centric scenario. The results show that—while relieving programmers from a significant burden—SQUIRREL achieves efficient utilization of storage areas, enabling energy savings independently of the storage technology.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Algorithms, Languages, Performance

1. INTRODUCTION

Early deployments of wireless sensor networks (WSNs) consisted of embedded devices that immediately communicated sensed data to the user [1]. Accordingly, the dominating design was characterized by a sense-and-send pattern, possibly with some local filtering. In this communication-centric setting, the storage capabilities of the devices play little role.

This strategy has proved to be not always successful, due to the difficulties in setting up energy-efficient, real-time collection networks [2]. Moreover, as deployments have grown in complexity, the amount of data harvested from the environment has started to outweigh the capabilities of present radio devices [3, 4]. WSNs have also entered fields where real-time data collection is not feasible. For instance, this is the case when nodes are attached to roaming entities [5, 6], or when deploying a base station is not possible [7, 8, 9].

A new breed of *storage-centric* sensing systems has emerged to tackle these issues [8, 10, 11, 3]. In this setting, storage operations tend to prevail over communication activity. This paradigm shift—favored by decreasing costs and increasing capacity of storage hardware—brings several advantages. It allows the system to perform batch data collection [11, 4], providing significant energy savings. It also enables delay-tolerant mobile applications [5, 6] and disconnected operations in static networks [7, 9], by providing means to retain data until the first upload opportunity arises.

Challenges. The programming fabric for storage-centric applications is largely missing. As we discuss in Section 2, WSN programmers are used to rely on file systems [12, 13] to store data on memory devices such as flash chips. However, the abstractions offered by WSN file systems are typically quite far from the application requirements, especially when in-network processing comes into play. This forces programmers to fill the gap "by hand", translating application-level functionality into low-level calls. This practice yields entangled implementations that are difficult to debug, re-use, and maintain. The issue resembles the mismatch in communication-centric designs between application requirements and low-level message-passing facilities [14], with file systems to play the role of the latter.

At system level, key to the efficient operation of a storage-centric system is deciding *where* to store data, e.g., whether to use the energy-efficient, yet limited main memory, or a more power-hungry, but larger external storage facility. Although there may be no choice due to the data size, it is often the case that different types of data fit multiple storage areas. These typically expose different size vs. energy trade-offs [15]. However, statically allocating data to storage areas is in general not possible, as data volumes may vary [4, 5, 7].

Dynamic allocation is a problem that application programmers are not keen on tackling, as it shifts their focus away from the application logic. Virtual memory systems [16, 17] exist, but they are mostly at the operating system level and incur severe performance penalties. Thus, programmers adopt the straightforward solution: using exclusively the largest storage area, even if inefficient from an energy standpoint.

Contribution and road-map. To address the issues above, we design and implement SQUIRREL[1], a storage-centric programming framework that *transparently* manages storage functionality. Our contribution is twofold:

- we design a stream-oriented, modular programming abstraction that cleanly decouples data processing from storage. As described in Section 3, programmers implement SQUIRREL programs by interconnecting *stream operators* that realize different storage policies, providing application-specific functionality when required. To ease adoption and leverage the existing code base, SQUIRREL's abstractions are reified in the C language.
- to leverage the size vs. energy trade-off of different storage areas, we map the storage problem to an instance of the "knapsack problem" [18], as we illustrate in Section 4. Based on the corresponding solution, the SQUIRREL run-time layer dynamically decides how to use different storage areas. Our decision procedure runs in only 20 ms in the most complex configuration we tested. We trigger its execution when the system detects a peak in memory consumption. Indeed, we recognize that this configuration is the one to consider for optimizing the overall performance.

We evaluate SQUIRREL along two key dimensions. Section 5 reports on how SQUIRREL impacts the programming activity, and discusses its generality. To this end, we consider three real-world applications representative of paradigmatic storage-centric scenarios, and compare SQUIRREL implementations against the original ones. We observe that SQUIRREL greatly simplifies the implementations, boosting code re-use and enabling easier maintenance.

In contrast, Section 6 investigates SQUIRREL's run-time performance. Micro-benchmarks show the limited memory and processing overhead. Experiments based on real-world data assess the effectiveness of our storage management scheme. As example, in batch data collection we show average reductions of 54.6% in the number of external storage operations by using main memory as additional storage area. A report from a prototype deployment completes the picture by demonstrating the use of SQUIRREL in a real scenario.

Section 7 compares our efforts against the current state of the art, and Section 8 concludes the paper.

2. MOTIVATION

We study the design and implementation of three real-world applications. Each of them represents a paradigmatic storage-centric scenario. The insights described here drive the design of SQUIRREL, illustrated in Section 3.

2.1 Batch Data Collection

Dutta et al. [11] observe that batching data and postponing communication provides many opportunities for optimization. Several real-world deployments leverage similar techniques to circumvent bandwidth limitations [3, 4].

A concrete example. Based on direct experience, we recognize that implementing the functionality for batch data collection is not necessarily trivial. For instance, in the *Torre Aquila* deployment [4], WSN nodes sense acceleration at several hundred Hz to assess the structural condition of a medieval tower. The end-user dynamically decides the duration and rate of sensing through a remote interface. This determines the data volumes at hand.

Due to bandwidth limitations, acceleration nodes iterate in a sense-compress-report loop. The acceleration sensor is queried at the desired rate and data immediately saved in blocks on an external FRAM [4] chip. Next, data is moved from external storage to main memory, each block is compressed, and then written back to external storage. Finally, compressed data is read from external storage, each block is divided into packets, and these are handed over to a collection protocol for transmission. The implementation of this functionality is one of the most complex in Torre Aquila.

2.2 Delay-tolerant Mobile Applications

WSN devices attached to mobile entities [5, 6] are likely to be only sporadically in contact with a collection station. Therefore, programmers must log data locally while awaiting an upload opportunity. Compression [19] and summarization [5] are used to make room for new data. Data replication is also applied [5] to increase the chances of delivering data to the user.

A concrete example. We are working on a typical example of mobile delay-tolerant application, called *BumpInto* [20]. Individuals carry WSN nodes to track their encounters and patterns of movement. Based on these data, social network experts can study the interactions among individuals.

We register a contact between two individuals when their WSN nodes are within radio range. We create a record to describe the encounter and save it on an external flash chip. The volume of data involved is difficult to predict, as it depends on the number and duration of contacts.

To achieve better utilization of storage space, we locally compute the social metrics of interest, e.g., the frequency of encounters. At every hour, we read the contacts recorded in the last six hours, compute some encounter summaries, and delete the contacts in the earliest hour. We replicate the summaries on nearby nodes using a gossip-based scheme [21]. A node uploads all summaries to a collection station whenever in contact with it.

[1]SQUIRRELs are known to be particularly concerned with "efficient" storage of their food supplies.

2.3 Disconnected Operations in Static Networks

Design simplifications motivate the use of static WSN devices working in absence of a collection station [7, 9]. Data may be retrieved using data mules [22], an intermittent long-range wireless link, e.g., a GPRS connection, or by recollecting the nodes. Programmers leverage local processing to deal with storage shortages and data redistribution to balance asymmetries in storage load [8].

A concrete example. In the *Klimat* [23] project, WSN devices are deployed on buoys in the Baltic sea to monitor environmental changes. Light and acceleration nodes are installed on the top of the buoy. The system is powered using rechargeable batteries fed by solar panels and a device that harvests energy from sea waves. When the available power suffices, data is transferred onshore using a GPRS device.

Due to unpredictable energy availability, the system is designed to work in a disconnected fashion. We store sensed data on a SD card as soon as it is gathered. Data compression is applied before upload operations to reduce the energy consumption due to the GPRS device. Different nodes may generate different amounts of data because of data-dependent variations in the compression ratios. Thus, we redistribute data among nodes similarly to EnviroStore [8].

2.4 Analysis

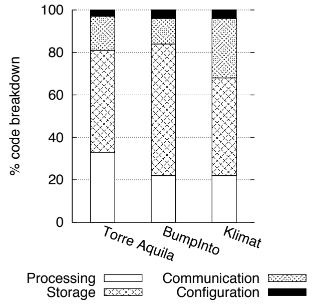


Figure 1: LOC breakdown depending on functionality for Torre Aquila, BumpInto, and Klimat.

We study the current implementations of the applications above to assess their flexibility. For instance, we note that in all cases the use of external storage is hardwired in the application code. In Torre Aquila, however, depending on the duration and rate of sensing, data may also fit in main memory. Similarly, depending on encounter frequency and upload opportunities, in BumpInto we may store data in main memory. Both implementations, however, cannot adapt to varying data volumes, and always use the more power-consuming external storage. The program binary image, however, is close to the limit on the nodes employed [24], which makes further optimizations difficult to achieve.

We also aim to understand how programmers devote their effort depending on the functionality. Figure 1 depicts a breakdown of lines of code (LOC) in each application, on a per-functionality basis. The chart only considers functionality implemented by application programmers, hence excluding OS mechanisms and general-purpose functionality. By considering this figure as an indication of programming effort, it appears that handling storage operations often represents a significant burden.

In Torre Aquila, storage operations are handled using TinyOS' `DirectStorage` interface [25], which provides `read()` and `write()` functions for specific memory blocks. Therefore, a significant amount of bookkeeping code is needed to bridge the gap between application-level functionality and `DirectStorage`. For instance, programmers must manually implement functionality to rewind the memory buffers while performing the necessary clean-ups, always doing so on the basis of physical memory addresses. Moreover, due to the need of interleaving storage operations with processing, e.g., during compression, the use of `DirectStorage` is often intertwined with application-specific functionality.

In both BumpInto and Klimat, storage operations are managed using the Coffee filesystem [12]. This provides traditional filesystem primitives such as `read`, `write`, and `append`. A filesystem API, however, is not sufficiently expressive to implement the functionality required. In BumpInto, we developed an ad-hoc intermediate layer to translate BumpInto storage operations into filesystem calls. This appears to be common practice in similar settings [12]. Nevertheless, storage operations are still interleaved with application-specific functionality, making the implementation fairly entangled. Similar observations apply to Klimat as well, as it uses the same file system API.

3. PROGRAMMING WITH SQUIRREL

To address the issues pointed out in the analysis above, we design SQUIRREL around two principles: *i)* processing/storage decoupling, and *ii)* transparent storage management. We take inspiration from data stream systems [26] to reify such design principles.

3.1 Abstraction

A SQUIRREL program is structured as a graph of *stream operators*. We provide six primitive types of operators, described in Figure 2. Nevertheless, programmers may straightforwardly implement additional custom operators as described next in Section 4.2. Every operator realizes a different storage policy and has an associated buffer space. Programmers supply application-specific functionality for data processing and to trigger the execution of operators. Data is represented as *items* of a predefined type whose internal format is application-dependent. Application-specific functionality operating on the same data must agree on its format.

We adopt this model for several reasons. First, our model belongs to the larger class of data flow paradigms that already demonstrated to be suited to WSN applications [27,28, 29,30,31], albeit rarely in storage-centric scenarios. Second, the stream abstraction helps factor out application-specific data processing from generic storage functionality. Third, structuring the application as subsequent manipulations of a data stream maps well to the step-wise execution of many storage-centric scenarios.

In the following, we revisit the design and implementation of the applications in Section 2 using SQUIRREL.

Torre Aquila. Figure 3 depicts a SQUIRREL implementation of batch data collection in Torre Aquila. The *Create* opera-

Operator	Description
Create	Generates a stream of items out of a programmer-provided function. It also timestamps every item using a predefined clock source.
DataSlide(X,Y)	Maintains a window of at most *X* items. After execution, it advances the window by *Y* items.
TimeSlide(X,Y)	Maintains a window of items with timestamps within the latest *X* time instants. It then advances the window by the items in the earliest *Y* time instants.
Move(X,destination)	Moves *X* data items from its local buffer to (1-hop) node *destination*. Receives moved items from other nodes and outputs them as a stream.
Join(X,Y)	Maintains two buffers of at most *X* and *Y* items, and outputs a single stream based on a programmer-provided function.
Execute(X,Y)	Executes a programmer-provided function for every item in its associated buffer.

Figure 2: SQUIRREL **stream operators and their functioning.**

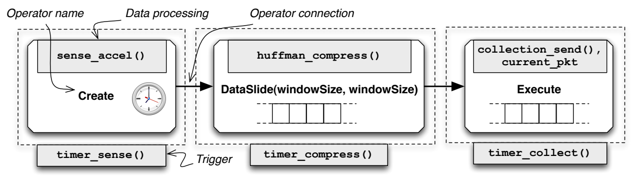


Figure 3: SQUIRREL **implementation of Torre Aquila. (Grey areas indicate application-specific functionality).**

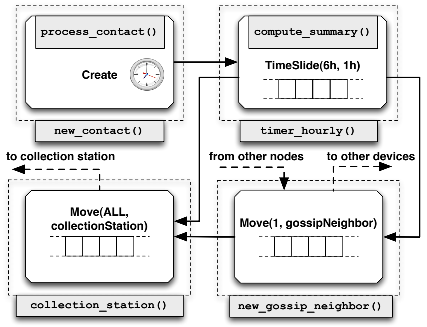


Figure 4: SQUIRREL **implementation of BumpInto.**

tor uses the `sense_accel()` function to create a stream of items carrying acceleration readings. The processing is triggered according to a `timer_sense()` function.

We use the *DataSlide* operator to implement the compression step. When `timer_compress()` triggers the operator, `windowSize` items are compressed using `huffman_compress()`. The window then advances by `windowSize` items to process the next block of readings.

We use the *Execute* operator to bind the stream to a collection protocol. When `timer_collect()` signals that reporting is to begin, `collection_send()` is executed for every item currently in the buffer. As multiple items may fit a single packet, a state variable `current_pkt` is carried across different executions to fill up a packet before transmission. Programmers may use state variables in all operators that accommodate programmer-provided processing.

BumpInto. As shown in Figure 4, we use *Create* to build an item upon detection of a contact with another node, using `new_contact()` as trigger and `process_contact()` to create the item. The *TimeSlide* operator buffers items created in the last six hours and, after execution, slides the window downstream by the items created in the earliest hour. We use this form of rolling computation to compute encounter summaries. The data processing is implemented in `compute_summary()`, and triggered every hour.

We argue that the *Move* operator captures most communication needs arising in storage-centric applications. In such scenarios, interactions are typically 1-hop. In BumpInto, for instance, we use *Move* to implement two functionality. Using `new_gossip_neighbor()` as trigger, we realize gossip-based replication [21]. The trigger implements the neighbor discovery functionality and decides whether to use the new neighbor to replicate data. If so, it makes *Move* execute to migrate `1` data item. On the receiver side, the same *Move* operator receives items from other devices, and hands them over to the following operators in the chain.

The last *Move* operator in the chain accumulates items until the collection station is in range. Whenever this occurs, the `collection_station()` function makes the second *Move* operator execute. This transfers `ALL` items in the buffer to the collection station. There, a corresponding *Move* operator receives the items, and an *Execute* operator dumps their content on a serial interface (not shown).

Klimat. We use two instances of *Create* to query different sensors, as Figure 5 shows. We use the *Join* operator to combine the two streams using the application-specific processing in `combine_accel_light()`. In this case, we use a "default" trigger (not shown). This makes *Join* execute when the maximum size for both buffers is reached. We use a default trigger also with *DataSlide*, which executes as soon as `windowSize` items are available[2].

In contrast to the use in BumpInto, here we use *Move* to implement balancing of storage load similarly to EnviroStore [8]. Using a dedicated function, nodes send periodic beacons carrying the current memory occupation. When we detect an unbalance, `storage_unbalance()` triggers the execution of *Move*. The number of items to move and the target neighbor are decided to avoid data "ping-pong" [8].

The operator chain in Klimat ends with an *Execute* operator that uses the GPRS device to send data onshore. This is triggered by a function that checks whether the current energy level is sufficient to operate the GPRS transceiver.

3.2 Language Constructs

Reifying a programming abstraction in a dedicated language may achieve greater conciseness and elegance. This comes at the cost of more difficult adoption and no re-use of the existing code-base. One may address these issues by embedding the abstraction within an existing language.

In the current version of SQUIRREL, we choose to adopt the latter approach by leveraging the C language. We believe that smoothing the waters for adoption by programmers is fundamental in a setting where the foundations are already available, but the programming fabric is largely missing.

As an example, Figure 6 reports the SQUIRREL code cor-

[2]Non-windowed operators using default triggers execute as soon as data is available.

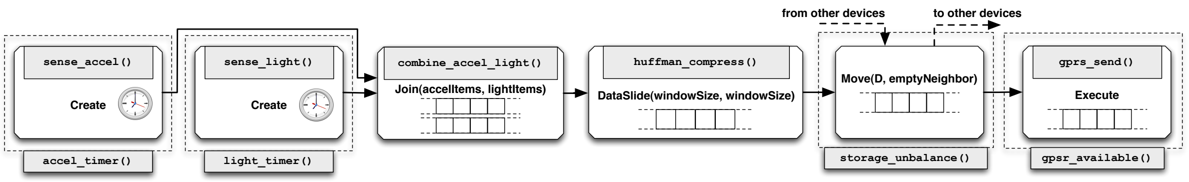


Figure 5: SQUIRREL **implementation of Klimat.**

```
1  /*-CREATE OPERATOR-*/
2  item_t sense_accel() {
3    item_t accel_data;
4    // Sensing from the accelerometer...
5    return accel_data; }
6  CREATE_OPERATOR(sense, sense_accel);
7  void timer_sense() {
8    TRIGGER_CREATE(sense);
9    // ... }
10 /*-DATASLIDE OPERATOR-*/
11 item_t huffman_compress(item_t* first_item,
12                         uint16_t window_size) {
13   item_t compressed_accel;
14   // Compressing window_size items...
15   return compressed_accel; }
16 DATASLIDE_OPERATOR(compress, huffman_compress,
17                    WINDOW_SIZE, WINDOW_SIZE);
18 void timer_compress() {
19   // ...
20   TRIGGER_DATASLIDE(compress); }
21 /*-EXECUTE OPERATOR-*/
22 item_t collection_send(item_t* item,void* current_pkt){
23   // Updating current_pkt and possibly sending data ...
24   return NULL; }
25 EXECUTE_OPERATOR(collect,collection_send,current_pkt);
26 void timer_collect() {
27   // ...
28   TRIGGER_EXECUTE(collect); }
29 /*-SETUP-*/
30 void setup() {
31   CONNECT(sense, compress);
32   CONNECT(compress, collect); }
```

Figure 6: SQUIRREL **code for Torre Aquila.**

responding to the operator chain in Figure 3. We make extensive use of standard C macros. For instance, `CREATE_OPERATOR` (line 6) instantiates a *Create* operator, taking as parameters the name of the operator instance (`sense`) and that of the application-specific sensing function (`sense_accel`). The macro `DATASLIDE_OPERATOR` (line 16) accepts two more parameters representing the size of the window and the number of items to slide after execution. Similar macros are available for all other types of operators.

Application-specific functionality exchange data with operators using a predefined `item_t` data type. To feed an operator with data, the functions simply return the data at the end of the processing (e.g., lines 5 and 15). Application-specific triggers use a dedicated macro to make an operator execute, which takes the name of the operator as parameter. For instance, `TRIGGER_CREATE` (line 9) causes the execution of `sense`, which is an instance of *Create*.

To configure the operator chain, programmers use a `CONNECT` macro (lines 31-32), which takes a $\langle source, sink \rangle$ pair of operator instances as parameters.

4. RUN-TIME SUPPORT

Data stream systems are most often designed to optimize the real-time operation, e.g., to maximize throughput [26]. In contrast, our objective is to optimize storage operations by exploiting the size vs. energy trade-offs of multiple storage areas. For instance, current WSN platforms often host a large storage facility, e.g., a flash chip, in addition to the limited main memory. These exhibit a size vs. energy trade-off that is expected to remain [15]. To improve energy consumption, we reduce the number of operations on flash by taking advantage of main memory whenever possible. Because of this, our work is also different from solutions that optimize the low-level operations depending on the storage technology at hand [10, 12, 13, 32]. Indeed, our goal is to avoid the use of external storage in favor of more energy-efficient, but possibly smaller, storage areas.

The kind of optimizations we aim at is difficult in the the general case. However, our stream-oriented abstractions limit the patterns of storage operations, enabling simple, yet effective techniques at system level. In this sense, although our approach resonates with memory hierarchies in mainstream computing, our solution differentiates from the latter in leveraging the characteristics of the data stream abstraction, rather than aiming at general-purpose functionality.

The following discussion considers two storage areas, main memory and one external facility. This is the common case in current WSN architectures. Multiple external storage areas with different size vs. energy trade-offs are also starting to appear [33]. Our approach requires straightforward extensions to take advantage of them.

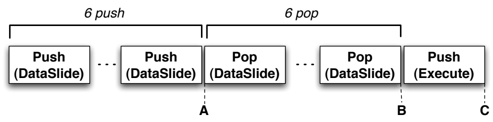
4.1 Storage Management

Identifying an efficient mapping of data to different storage areas is not trivial, especially if data volumes change unpredictably. We discuss next an illustrative example.

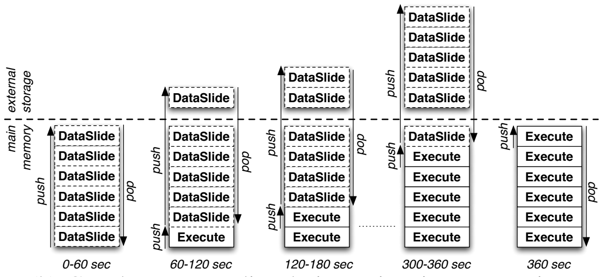
Example. Consider a simplified version of the program in Figure 3. Say *Create* generates one item every 10 sec. *DataSlide* buffers up to six items and then applies compression, thus outputting one item per minute. Every six minutes, *Execute* hands six items over to the collection protocol.

Figure 7(a) illustrates the storage operations performed during a one minute execution of this application. These are either push or pop operations on a specific buffer, and include: *i)* six push operations in *DataSlide* as sensed data arrive (point A); *ii)* six pop operations when *DataSlide* compresses data (point B); *iii)* one push operation in *Execute* to store the item coming from *DataSlide* (point C). Moreover, every six minutes the system performs six pop operations in *Execute* to hand data over to the collection protocol. Say 60 bytes in main memory are available as storage area, and a single item occupies 10 bytes. Even in this simple setting, we may apply different storage policies.

Using a greedy policy, we store every item in main mem-



(a) Storage operations during a one minute execution.



(b) Greedy storage policy during a six minute execution.

Figure 7: Storage management example.

ory when possible and resort to external storage otherwise. The memory occupation thus evolves as shown in Figure 7(b). From 0 to 60 sec, *DataSlide* pushes six items. At 60 sec, *DataSlide* pops the same six items, *Execute* then finds the main memory empty, and it stores there the item it receives from *DataSlide*. From 60 sec to 120 sec, because of the item stored earlier by *Execute*, *DataSlide* finds space in main memory for all but the last item. This is thus saved on external storage. The same processing occurs from 120 sec to 180 sec, when *DataSlide* fits in main memory all but the last two items. Execution continues this way until the fifth minute, when *DataSlide* can store only one item in main memory. At 360 sec, *Execute* pops all items in its buffer, and main memory is newly empty. Using this strategy, the system performs a total of 30 operations on external storage.

Storage strategy. In the example above, a more efficient strategy is to reserve main memory for buffers with higher frequency of push/pop operations. By using main memory only for *DataSlide*, the number of operations on external storage becomes only 12, a 60% reduction.

To achieve this functionality in the general case, we map the problem at hand to the "knapsack problem" [18]. We are given a set of objects with associated weight w_i and value p_i. We must determine which objects to include in a collection (knapsack) so that: *i)* the total value of objects in the collection is maximized, and *ii)* the total weight is less than a limit W. In our mapping, the knapsack is the MCU's *main memory*, and an object is an *item* in a buffer. The object's weight is the *size* of the item in memory, and its value is the *frequency* of push/pop operations on the buffer it belongs to.

Based on a snapshot of the memory occupation at a given point in time, a solution to the knapsack problem determines a combination of "preferred" items that fit in main memory and belong to buffers with high frequency of push/pop operations. From that point on, we use the greedy strategy explained earlier for preferred items, and we allocate non-preferred items immediately on external storage. The former is needed in case the data volumes change w.r.t. the inputs used to run the knapsack algorithm.

Dynamic behavior. To decide *when* to run the knapsack algorithm, we observe that the evolution of memory occupation is likely to follow specific patterns, exemplified in Figure 8. Over time, the maximum of memory occupation keeps growing until the node has an opportunity to flush all items. For instance, this happens when reporting finishes in batch data collection, a mobile node comes in contact with a collection station, or a data mule passes by in a disconnect scenario. We also note that most often there is just one operator that is mainly responsible for the maximum, e.g., operator *A* in Figure 8. This is typically at the end of the chain, and accumulates items awaiting an upload opportunity.

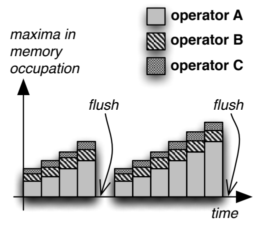


Figure 8: Typical evolution of maxima in memory occupation.

Leveraging these observations, we choose to run the knapsack algorithm when the system reaches a new *peak* in memory consumption in a given period, called "knapsack period". This allows the knapsack algorithm to obtain a memory snapshot where all operators that contribute to the peak appear. Implicitly, this allows to optimize how this worst-case situation creates. For instance, in Figure 7(b) the execution at 360 sec is the one ultimately taken into account. This shows that only the six *DataSlide* items should be allocated in main memory. Their frequency of push/pop operations is indeed higher than *Execute* items, and they fill up the main memory.

Keeping track of memory peaks only in the knapsack period allows the system to adapt to varying data volumes. If these changes affect memory consumption, the system may reach different peaks, not necessarily greater than in previous executions. Therefore, the system should eventually forget about past peaks. The length of the window is decided based on the application dynamics, as we exemplify in Section 6.

Knapsack algorithm. We adapt a well-known approximate algorithm [18]. First, the algorithm sorts the items in decreasing order of value per unit of weight p_i/w_i. Then, it inserts the items into the knapsack starting with the one with highest p_i/w_i, until there is no space to fit more items.

By definition, all items belonging to the same buffer have the same p_i and w_i. Thus, we can carry out the ordering step on a per-buffer basis. For the same reason, insertion in the knapsack can occur at buffer granularity, by computing how many items of the same buffer fit into the available space. These observations yield an algorithm that—while deciding on the allocation of individual data items—scales with the number buffers, which are expected to be much fewer.

4.2 Implementation

Our current implementation runs atop the Contiki [34] operating system and targets TMote-like [24] devices.

Operators are implemented as Contiki processes. This allows using Contiki events to trigger their execution and to realize the flow of data items. When an operator outputs an item, this is processed by a router module that determines the target operator based on the current connections. Delivery of the item to the target operator occurs by posting an event to the corresponding process.

The router module is also responsible for the storage strategy, and thus determines where to store items. To manage storage operations, we re-implemented Contiki's `LIST` library to allow entries in the list to be stored indifferently on main memory or external storage. To implement this functionality, we add a flag to every item to indicate where to find the next entry in the list. For efficiency reasons, we also keep pointers to the first and last entry in the list. We use the Coffee [12] filesystem to access the external storage, mapping every buffer to a separate file.

Programmers can straightforwardly implement custom operators if required. To do so, they leverage the API of the `LIST` library to manage storage operations. Before every such operation, the operator must ask the router module what storage area to use to store the item. This decision is taken based on the observed frequency of push/pop operations in the last knapsack period, which is periodically reported to the router module by every operator. Based on this information, the router module applies our storage strategy identically to the predefined operators.

To configure the run-time layer, programmers specify the fraction W of main memory for SQUIRREL operations, and the knapsack period.

5. INVESTIGATING SQUIRREL PROGRAMMING

We aim at investigating the simplifications brought by SQUIRREL to the development of storage-centric applications, while discussing its general applicability. It is notoriously difficult to obtain objective indications on the effectiveness of a programming framework. We leverage the body of work on software metrics [35] to tackle this challenge. Moreover, in our case, embedding a new abstraction within an existing language simplifies the problem. It helps isolate the advantages brought by the newly introduced abstractions, as the rest of the framework remains the same.

Throughout the discussion, we consider the original implementations of BumpInto and Klimat, described in Section 2, against their SQUIRREL counterparts, illustrated in Section 3. To achieve a fair comparison, in the case of Torre Aquila we port the existing implementation to Contiki. We refer to these implementations as *base* implementations. Our analysis considers only the portions of code that application programmers are to implement, hence excluding OS-provided functionality and general-purpose mechanisms.

Our analysis covers different facets, as described next.

5.1 Coupling

Methodology. Based on the seminal work by Stevens et al. [36], seven types of coupling, summarized in Figure 9, are commonly recognized between software modules or functions. It is generally acknowledged that the tightest is the coupling, the more difficult is debugging, re-using, and maintaining the implementations. We determine the types of coupling we observe in base and SQUIRREL implementations.

Type	Description
Content (tight)	One module relies on the internal working of another. Changing one module requires changes in the other as well.
Common	Two or more modules share some global state, e.g., a variable.
External	Two or more modules share a common data format.
Control	One module controls the flow of another, e.g., passing information that determine how to execute.
Stamp	Two or more modules share a common data format, but each of them uses a different part with no overlapping.
Data	Two or more modules share data through a typed interface, e.g., a function call.
Message (loose)	Two or more modules share data through an untyped interface, e.g., via message passing.

Figure 9: Types of coupling between software modules.

	Content	Common	External	Control	Stamp	Data	Message
Torre Aquila - base	-	Yes	Yes	-	Yes	Yes	-
Torre Aquila - SQUIRREL	-	-	Yes	-	-	-	Yes
BumpInto - base	Yes	Yes	Yes	Yes	-	Yes	Yes
BumpInto - SQUIRREL	-	-	Yes	Yes	-	-	Yes
Klimat - base	Yes	Yes	Yes	Yes	Yes	Yes	-
Klimat - SQUIRREL	-	-	Yes	Yes	-	-	Yes

Figure 10: Coupling in base and SQUIRREL implementations.

Results. Figure 10 illustrates the results of our analysis. In the applications we consider, using SQUIRREL removes several types of coupling found in base implementations. We maintain that this is due to the separation between data processing and storage we enable in SQUIRREL. The storage policies in our operators can be (re)used orthogonally to the data processing involved, generally loosening the coupling.

In SQUIRREL implementations, *External* coupling appears between different operators, in that programmer-provided functionality that operate on the same data items must agree on their internal format. Items flowing through different operators represent a form of *Message* coupling, as they resemble messages passed through an un-typed interface. In addition, SQUIRREL programs possibly show *Control* coupling whenever a trigger passes parameters to drive the execution of an operator, e.g., as in BumpInto between `new_gossip_neighbor()` and *Move*, shown in Figure 4. Note that, however, these types of coupling were already present in the base implementations.

5.2 Complexity

Methodology. The number of lines of code (LOC), the number of variable declarations, and the number of functions are generally considered as indications of a program's complexity. It is also observed that complexity is a function of the number of states in which the program can find itself [35]. A state here is any *possible* assignment of values to the program variables. Thus, the number of states must be computed by looking at the different combinations of values assumed by

Application	LOC	Variable declarations	Functions	Per-function states	
				Average	StdDev
Torre Aquila - base	4523	32	43	1312.31	1031.82
Torre Aquila - SQUIRREL	-35%	-66%	-51%	-61%	-89%
BumpInto - base	3211	24	37	913.56	532.72
BumpInto - SQUIRREL	-54%	-66%	-48%	-41%	-69%
Klimat - base	2341	43	54	1123.76	341.41
Klimat - SQUIRREL	-42%	-68%	-37%	-43%	-42%

Figure 11: Code complexity in base and SQUIRREL implementations.

variables during every possible execution.

To carry out this analysis, we use SATABS [37], a verification tool for C programs. SATABS is designed for off-line verification of C programs against user-provided assertions. To do so, it searches through the relevant program executions to check whether the assertion always holds. At the end of the process, SATABS returns the number of different states it explores in the program.

Using a specific configuration, it is possible to force SATABS to explore *all* program executions. If the procedure terminates, SATABS returns the total number of distinct states of the program. We use SATABS on a per-function basis, implementing empty stubs to replace code that we cannot process with SATABS, e.g., hardware drivers.

Results. We illustrate the results of the analysis in Figure 11. Using SQUIRREL enables significant reductions in all metrics. The reduction in LOC for BumpInto and Klimat is comparable to the fraction of code devoted to storage operations illustrated in Figure 1. SQUIRREL spares most storage functionality by transparently managing the corresponding operations. Because of the same reason, the number of variable declarations and functions decreases as well. Essentially, in SQUIRREL implementations these cater only to data processing. Based on the number of per-function states, it also appears that the individual functions are simpler and more homogeneous. The latter aspect is observed as the standard deviation in the number of per-function states decreases.

5.3 Generality

SQUIRREL's applicability extends in storage-centric scenarios beyond the applications we consider. For instance, SQUIRREL is generally amenable to disconnected applications that use replication and redistribution schemes [9, 7]. These are often composed of a trigger condition and some storage functionality, which can be cleanly implemented using the *Move* operator, further highlighting its generality.

Another example deals with windowed operators. In our applications, we use them for compression and summarization. Alternatively, they can be applied to reorder batches of data based on application-specific priority policies [5]. To do so, programmers implement the necessary policy enforcement in the data processing functions, while the underlying storage operations are still managed by SQUIRREL.

On the other hand, we believe that one of SQUIRREL's limitations lies in the unidirectional flow of data. It is thus difficult to implement functionality to process the *same* data in multiple iterations [38]. One might connect an operator to itself, creating a loop in the chain. It is unclear, however, how the trigger should drive the execution of such operator.

Functionality	Data memory	Program memory
Run-time core	452 bytes	6.9 Kbytes
Create operator	12 bytes	196 bytes
DataSlide operator	22 bytes	328 bytes
TimeSlide operator	28 bytes	532 bytes
Join operator	32 bytes	466 bytes
Move operator	48 bytes	1.1 Kbytes
Execute operator	20 bytes	598 bytes
Data item	4 bytes	-
Operator connection	4 bytes	-

Figure 12: SQUIRREL memory overhead.

SQUIRREL's data stream model may also be ill-suited to applications that require to perform complex searches into structured data. A database model may be a better fit for this task. There exist efficient database implementations running on the external storage facilities of WSN nodes [32]. If required, these may be integrated into a SQUIRREL program using an *Execute* operator.

6. RUN-TIME PERFORMANCE

We first investigate SQUIRREL memory and processing overhead through micro benchmarks. Next, we study the performance of our storage management scheme based on real-world data. Finally, we discuss deployment experiments.

6.1 Micro Benchmarks

Memory overhead. Figure 12 summarizes the memory overhead of different SQUIRREL functionality. All values are reasonably limited. The only fixed cost is the run-time core. This includes the Coffee filesystem we use to access the external storage, which accounts for about 5 Kbytes in program memory alone. The *Move* operator requires linking some network-level functionality that would not be used otherwise. The 4 bytes required for every data item are used to implement the `LIST` library described in Section 4.2.

In our implementation, non-used operators bear no effect on memory consumption. Therefore, the total memory overhead depends on the specific application. For instance, this figure amounts to 8.1 Kbytes (program) and 514 bytes (data) in Torre Aquila, 9.8 Kbytes (program) and 604 bytes (data) in BumpInto, and 9.9 Kbytes (program) and 606 bytes (data) in Klimat. Compared to the base implementations, in the worst case the size of the program binary image is only about 1.5 Kbytes larger when using SQUIRREL, while the occupation in data memory is always comparable.

Processing overhead. The execution of the knapsack algorithm is the main source of processing overhead. To quantify this aspect, we use the MSPSim emulator [39]. For easier interpretation, we use a synthetic setting where all data items are of the same size. We assign weights to items in a way to create a worst-case situation for sorting.

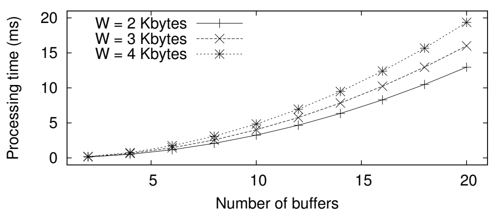


Figure 13: Processing time of the knapsack algorithm.

Figure 13 depicts the trends at stake. As expected, the processing time grows quadratically with the number of buffers. The slight increase with the size W of main memory devoted to storage operations is because larger Ws accommodate more items, and the processing for preferred items is slightly more complex than for non-preferred ones. Nevertheless, the absolute values in Figure 13 are limited, also considered the size of the input data. We believe that the number of buffers involved overestimates the needs of typical storage-centric applications. The applications in Section 3, for instance, require at most 5 buffers.

6.2 Storage Management

We consider data collected in previous deployments of Torre Aquila, BumpInto, and Klimat. Based on these, we replay the application execution while counting the operations performed on external storage. This way, factors outside the operator chain, e.g., the detection of nearby devices in BumpInto, are the same in all cases.

We compare the performance of SQUIRREL against: *i)* the base implementations, described in Section 2; *ii)* the greedy allocation strategy described in Section 4.1, which we call SQUIRREL-greedy; and *iii)* the theoretical lower bound in the number of external storage operations. We use the Cooja simulator and MSPSim [39] to run *i)*, *ii)*, and SQUIRREL. To compute *iii)*, we implement a Java program that explores all possible executions in a "brute-force" manner.

We allocate $W = 3$ Kbytes for storage operations in main memory, a value that avoids memory overflows in the implementations we consider. In general, W should be set to exploit all memory left by other functionality running on the node. To understand the impact of this value, we also discuss results with $W = 2$ Kbytes. Similarly, we also study the effect of the knapsack period by varying its setting when precise apriori information would be unavailable to decide on a specific value.

6.2.1 *Torre Aquila*

Setting. We consider five weeks of acceleration data sensed at different nodes. During this time, the end users were submitting week-long tasks with the sampling phase set to 30 secs and varying sampling frequency.

A single acceleration sample is 12 bits in size. The *Create* operator in Figure 3 batches one second of acceleration data in every data item, identically to the base implementation. We set `windowSize` in *DataSlide* to accommodate a 500 Hz sampling frequency for 1 min (the limit of the deployed system [4]). The knapsack period is set to the data collection period, which is known. Thus, every iteration of the sense-compress-report loop is considered separately.

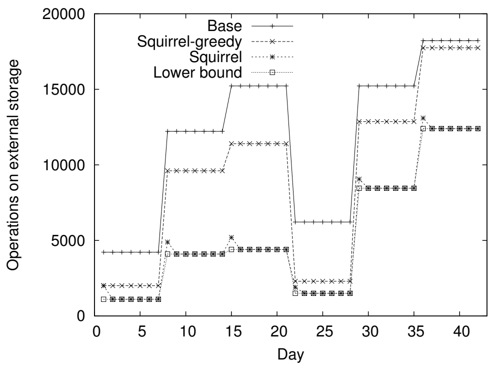


Figure 14: Torre Aquila: number of external storage operations over time. ($W = 3$ Kbytes).

Results. Figure 14 reports the number of operations on external storage throughout the data set, aggregated on a daily basis. The trend is mainly driven by the sampling frequency of the sensing task, which changes weekly and ultimately determines the amount of data to store.

The chart shows that SQUIRREL improves on both base and SQUIRREL-greedy. Throughout the data set, SQUIRREL reduces the external storage operations by factor of 54.6% (42.32%) w.r.t. the base (SQUIRREL-greedy) implementation. The gains over the former stem from using main memory as additional storage area. On the contrary, the use of external storage is hard-wired in the base implementation. Instead, the improvements w.r.t. the greedy scheme are due to recognizing that the buffer in *DataSlide* is used most often. Using main memory for it is thus more efficient.

The behavior of our storage allocation scheme approaches the lower-bound but at the beginning of a new task, as the small, weekly spikes in Figure 14 testify. Our solution needs to observe at least one iteration of the new task to obtain information on the data volumes. During this iteration, SQUIRREL runs with the earlier configuration. This may be no longer efficient, e.g., because of frequency-dependent changes in the compression ratios. The off-line scheme anticipates these changes. Nevertheless, after recognizing the new setting, SQUIRREL approaches the lower-bound.

Using $W = 2$ Kbytes causes an average 52.32% (58.12%) reduction in the improvements over the base (SQUIRREL-greedy) implementation. Smaller Ws limit the the use of main memory as additional storage area. However, given the advantages brought to application development and the limited memory/processing overhead imposed by SQUIRREL, these improvements are still valuable.

6.2.2 *BumpInto*

Setting. We consider a ten day deployment of BumpInto at our institution [20]. An average of 18 persons participated to the experiment by carrying one TMote-like node in their pockets between 9 AM and 6 PM, yielding a total of 178

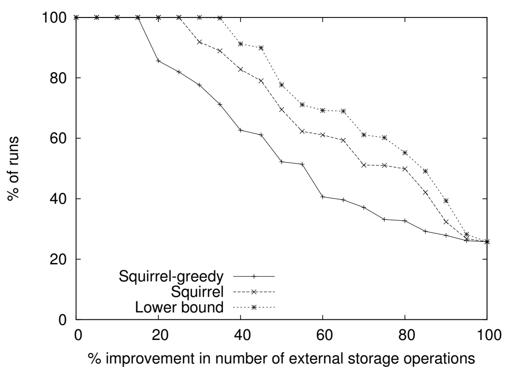


Figure 15: BumpInto: complementary CDF of % improvement in external storage operations over base implementation, vs. percentage of runs. ($W = 3$ Kbytes).

runs. These persons belong to our research group, so they have frequent encounters. We deployed a collection station at the office of one of the participants.

An encounter record is 42 bytes in size. Initially, we set the knapsack period to 30 min. Based on previous experiments with BumpInto, we recognize that this is the average duration of pair-wise interactions in our group.

Results. Compared to Torre Aquila, the application dynamics in BumpInto are more irregular. They depend on random factors such as the frequency and duration of contacts. There is thus high variability in the data volumes across nodes.

Figure 15 depicts the complementary cumulative distribution function (CDF) representing the percentage improvement in the number of external storage operations over the base implementation, against the percentage of runs showing such improvement. For instance, it shows that in 40% of the runs SQUIRREL-greedy saves at least 60% of the external storage operations performed by the base implementation.

The chart shows that SQUIRREL provides greater improvements than SQUIRREL-greedy. SQUIRREL recognizes that operations in *TimeSlide* and in the intermediate *Move* operator of Figure 4 are the most frequent. On the other hand, both solutions save *all* external storage operations in about 25% of the runs. This happens at nodes that are often nearby the collection station. They have frequent upload opportunities, even before the use of external storage becomes necessary.

Figure 15 also illustrates how SQUIRREL improvements are always within 9% from the lower-bound. This gap is caused by the unpredictability of the data volumes. Again, the off-line scheme predicts changes in data volumes, and adapt the allocation strategy before they happen. In contrast, SQUIRREL needs to observe these changes to acquire the information necessary to determine a better allocation strategy.

Setting $W = 2$ Kbytes reduces the improvements over the base (SQUIRREL-greedy) implementation by a factor of 54.11% (59.87%). Moreover, setting the knapsack period to ±25% of the initial value yields a worst-case performance degradation of only 7.22% w.r.t. to both base and SQUIRREL-greedy implementations. A reasoning as in the case of Torre Aquila applies. Despite the reduced performance, the simplification to the development activity and the limited processing/memory overhead still make SQUIRREL a viable choice.

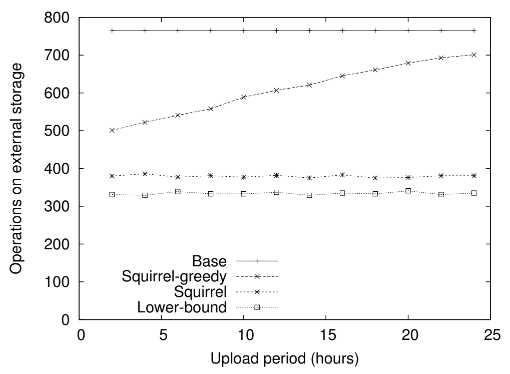


Figure 16: Klimat: number of external storage operations against upload period. ($W = 3$ Kbytes).

6.2.3 Klimat

Setting. We consider acceleration and light data taken during a one-day pre-deployment experiment. Storage balancing and energy harvesting were not used in this installation. Thus, we miss traces of data transfers between nodes, and we cannot determine the period of uploads onshore. To remedy this, we simulate the presence of 4 nodes on a buoy that process different subsets of our data set, mimicking the target deployment setting. These nodes are in the same communication range. We artificially vary the upload period to study its impact on the performance.

The knapsack period is initially set to the average upload period, for which some indications are expected to be available before deployment. As Klimat also uses a Huffman-based compression scheme, we set the other parameters as in Torre Aquila.

Results. Figure 16 illustrates that our storage management scheme almost halves the number of operations on external storage compared to the base implementation. The behavior of the latter is constant because the use of external storage is hardwired in the application code. Small variations are observed in SQUIRREL because of data-dependent variations in the compression ratios, which affect the data volumes.

SQUIRREL improves by an average factor of 35.67% over SQUIRREL-greedy, whose behavior is roughly linear with increasing upload periods. As uploads are less frequent, more data accumulates at the end of the chain, where storage operations are less frequent. Being oblivious to such information, SQUIRREL-greedy may use main memory for items that are removed only when an upload occurs. This causes more frequently used items to be allocated to external storage. This situation becomes more frequent as uploads are more rare.

In this setting, SQUIRREL is within 18% from the lower bound, on average. The gap is still due to the ability of the off-line scheme to predict changes in the data volumes. In Klimat, the effect of this is evident at system start-up, where SQUIRREL uses the greedy strategy because of lack of infor-

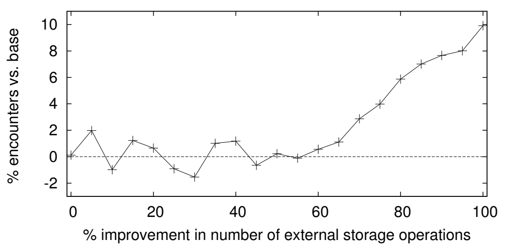


Figure 17: BumpInto: tracked encounters vs. improvements in external storage operations. ($W = 3$ Kbytes).

mation. About 64% of the gap from the lower-bound is due to decisions taken during the first upload iteration.

Using $W = 2$ Kbytes brings an average 47.65% (53.42%) reduction of the gains over the base (SQUIRREL-greedy) implementation. Moreover, setting the knapsack period to ±25% of the initial value corresponds to a worst case 5.46% performance penalty w.r.t. to both base and SQUIRREL-greedy implementations. This performance is still worthwhile, for reasons already discussed for Torre Aquila and BumpInto.

6.3 Deployment Experiments

We complete the evaluation by assessing the effectiveness of SQUIRREL in a prototype BumpInto deployment.

Setting. We consider a setting similar to that of BumpInto in Section 6.2. This time, however, 9 persons participate to the experiment, each carrying *two* WSN nodes. This allows us to run the SQUIRREL implementation alongside the base one. We use two non-overlapping radio channels, where we experimentally verify the absence of external interference.

We collect a total of 54 runs and look at the number of encounters tracked. In our setting, this figure is expected to be about the same in the two implementations.

Results. Figure 17 shows a relation we found between the improvements in number of external storage operations compared to the base implementation, and the number of encounters tracked. After a 60% improvement, SQUIRREL increasingly tracks more encounters than the base implementation. This reaches a 10% improvement on nodes that do not use external storage, for reasons explained in Section 6.2.

We conjecture that a possible cause for this behavior may be a reduction in the contention on the SPI bus due to fewer external storage operations. On TMote-like nodes, the bus is shared between radio and external storage. To arbitrate the bus, Coffee disables the radio interrupts when operating on external storage. This may cause some packet losses if, for instance, two packets are received in a short time while interrupts are disabled. The later packet over-writes the earlier one without the application knowing about it.

We can conclude that in this setting SQUIRREL not only simplifies the programming activity, but may also provide some improvements in the application performance.

7. RELATED WORK

The abstractions in SQUIRREL are influenced by data stream systems [26]. Data streams have also been applied to process sensor data [40] and to integrate WSNs in larger infrastructures [41]. In SQUIRREL, we leverage this model to decouple data processing from storage and to identify reusable storage policies.

Dataflow models for WSN programming have been proposed previously [27, 28, 29, 30, 31]. In most cases, the abstractions allow to process one data item at the time. This may be ill-suited to storage-centric applications, which often need to process data in batches. Eon [28] focuses on adapting the application processing to energy availability. More generally, Pixie [27] provides a framework to adapt to changes in the availability of resources, e.g., bandwidth and energy. In WaveScope [29], Flask [30], and ATaG [31], the dataflow model is essentially used to mask distribution. In contrast, our focus is to decouple data processing from storage operations, and to abstract the latter.

Halfway between SQUIRREL and file system abstractions we find solutions providing general-purpose abstractions to access external storage, along with technology-specific implementations [10, 32, 42]. Storage operations are explicit. SQUIRREL abstractions, instead, are conceived to hide storage operations. At system level, our focus is to avoid the use of external storage, rather than designing an efficient way to do it. Nevertheless, we may leverage these solutions as back-end for our run-time layer, e.g., by implementing storage operations on buffers using Capsule's *queue* object [10].

As already mentioned, file system abstractions [12, 13] are currently the predominant choice to handle operations on external storage. This imposes a significant burden on the programmer's shoulders. In SQUIRREL, we raise the level of abstraction to a point where storage operations are no longer visible, and use a file system as interface to external storage.

8. CONCLUSION

We presented SQUIRREL, a stream-oriented programming framework for storage-centric WSNs. SQUIRREL alleviates the programming burden by decoupling data processing from storage, and by transparently handling the latter. At system level, SQUIRREL exploits the size vs. energy trade-offs of multiple storage areas, providing technology-independent energy savings. We showed that SQUIRREL implementations are easier to understand, debug, and maintain. We also demonstrated significant savings in the number of operations on the more energy-consuming storage areas, at the price of limited memory/processing overhead.

Acknowledgements. The author thanks Fredrick Österlind, Kay Römer, and Thiemo Voigt for the feedback received on early versions of the paper, as well as the shepherd, Matt Welsh, for his advice in preparing the final version. This work was financed by SSF, Swedish Foundation for Strategic Research, and by VINNOVA, the Swedish Agency for Innovation Systems. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, under EU-FP7 contract number FP7-2007-2-224053.

9. REFERENCES

[1] K. Martinez, J. K. Hart, and R. Ong, "Environmental sensor networks," *Computer*, vol. 37, no. 8, 2004.

[2] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *Proc. of the* 14^{th} *Int. Wrkshp. on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[3] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[4] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corrà, M. Pozzi, D. Zonta, and P. Zanon, "Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment," in *Proc. of the* 8^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[5] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet," *SIGPLAN Not.*, vol. 37, no. 10, 2002.

[6] R. K. Ganti, P. Jayachandran, T. Abdelzaher, and J. Stankovic, "SATIRE: a software architecture for smart AtTIRE," in *Proc. of the* 4^{th} *Int. Conf. on Mobile Systems, Applications and Services (MOBISYS)*, 2006.

[7] L. Luo, Q. Cao, C. Huang, L. Wang, T. Abdelzaher, J. Stankovic, and M. Ward, "Design, implementation, and evaluation of enviromic: A storage-centric audio sensor network," *ACM Trans. Sen. Netw.*, vol. 5, no. 3, 2009.

[8] L. Luo, C. Huand, T. Abdelzaher, and J. Stankovic, "EnviroStore: A cooperative storage system for disconnected operation in sensor networks," in *Proc. of the* 26^{th} *Int. Conf. on Computer Communications (INFOCOM)*, 2007.

[9] Y. Yang, L. Wang, D. K. Noh, H. K. Le, and T. Abdelzaher, "Solarstore: enhancing data reliability in solar-powered storage-centric sensor networks," in *Proc. of the* 7^{th} *Int. Conf. on Mobile Systems, Applications, and Services (MOBISYS)*, 2009.

[10] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy, "Capsule: An energy-optimized object storage system for memory-constrained sensor devices," in *Proc. of* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, 2006.

[11] P. Dutta, D. Culler, and S. Shenker, "Procrastination might lead to a longer and more useful life," in *Proc. of* 6^{th} *Wrkshp. on Hot Topics in Networks (HotNets-VI)*, 2007.

[12] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the Coffee file system," in *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[13] H. Dai, M. Neufeld, and R. Han, "Elf: an efficient log-structured flash file system for micro sensor nodes," in *Proc. of* 2^{nd} *Int. Conf. on Embedded networked sensor systems (SenSys)*, 2004.

[14] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, 2010. To appear. Available at: `www.sics.se/~luca/papers/mottola10programming.pdf`.

[15] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," in *Proc. of the* 5^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2006.

[16] L. Gu and J. Stankovic, "T-Kernel: Providing reliable OS support to wireless sensor networks," in *Proc. of the* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, 2006.

[17] A. Lachenmann, P. J. Marrón, M. Gauger, D. Minder, O. Saukh, and K. Rothermel, "Removing the memory limitations of sensor networks with flash-based virtual memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, 2007.

[18] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Springer, 2004.

[19] C. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proc. of the Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, 2006.

[20] A. Dunkels, L. Mottola, N. Tsiftes, F. Österlind, J. Eriksson, and N. Finne, "Implicit announcements: Re-thinking the use of broadcast in mobile sensor networks," tech. rep., SICS, 2009.

[21] A. M. Kermarrec and M. van Steen, "Gossiping in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, 2007.

[22] R. Shah, S. Roy, S. Jain, and W. Brunette, "Data mules: Modeling a three-tier architecture for sparse sensor networks," in *Proc. of the* 1^{st} *Int. Wrkshp. on Sensor Network Protocols and Applications*, 2003.

[23] T. Voigt, F. Österlind, N. Finne, N. Tsiftes, Z. He, J. Eriksson, A. Dunkels, U. Bamstedt, J. Schiller, and K. Hjort, "Sensor networking in aquatic environments - experiences and new challenges," in *Proc. of the* 1^{st} *Int. Wrkshp. on Practical Issues in Building Sensor Network Applications*, 2007.

[24] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *Proc. of the* 5^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[25] TinyOS Community Forum, "TinyOS TEP 128 - Platform Independent Non-Volatile Storage Abstractions." `www.tinyos.net/tinyos-2.x/doc/html/tep128.html`.

[26] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. of the* 21^{st} *Symp. on Principles of Database Systems*, 2002.

[27] K. Lorincz, B. Chen, J. Waterman, G. Werner-Allen, and M. Welsh, "Resource aware programming in the Pixie OS," in *Proc. of the* 6^{th} *Int. Conf. on Embedded Network Sensor Systems (SenSys)*, 2008.

[28] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. Corner, and E. Berger, "Eon: A language and runtime system for perpetual systems," in *Proc. of the* 5^{th} *Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, 2007.

[29] R. Newton, G. Morrisett, and M. Welsh, "The Regiment macroprogramming system," in *Proc. of the* 6^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[30] G. Mainland, G. Morrisett, and M. Welsh, "Flask: Staged functional programming for sensor networks," in *Proc. of the* 13^{th} *Int. Conf. on Functional Programming*, 2008.

[31] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The Abstract Task Graph: A methodology for architecture-independent programming of networked sensor systems," in *Workshop on End-to-end Sense-and-respond Systems (EESR)*, 2005.

[32] S. Nath and A. Kansal, "FlashDB: Dynamic self-tuning database for NAND flash," in *Proc. of the* 7^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[33] Libelium Inc., "WaspMote." `www.libelium.com`.

[34] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - A lightweight and flexible operating system for tiny networked sensors," in *Proc. of* 1^{st} *Wkshp. on Embedded Networked Sensors*, 2004.

[35] R. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2001.

[36] W. Stevens, G. Myers, and L. Constantine, "Structured design," *Classics in software engineering*, 1979.

[37] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[38] S. Nath, "Energy efficient sensor data logging with amnesic flash storage," in *Proc. of the* 8^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[39] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "COOJA/MSPSim: Interoperability testing for wireless sensor networks," in *SIMUTools*, 2009.

[40] J. Gama and M. Gaber, *Data Stream Processing in Sensor Networks*. Springer, 2007.

[41] D. Abadi, W. Lindner, S. Madden, and J. Schuler, "An integration framework for sensor networks and data stream management systems," in *Proc. of the* 30^{th} *VLDB Int. Conf.*, 2004.

[42] C. Sadler and M. Martonosi, "DALi: A communication-centric data abstraction layer for energy-constrained devices in mobile sensor networks," in *Proc. of the* 5^{th} *Int. Conf. on Mobile Systems, Applications, and Services (MOBISYS)*, 2007.